UNIVERSIDAD DE MÁLAGA
E.T.S.I. TELECOMUNICACIÓN

Tablas de dispersión

Programación modular

(1º de Ingeniería de Telecomunicación)

Tablas de dispersión

Es posible encontrar implementaciones de la clase Tabla que ofrezcan un algoritmo de búsqueda cuya complejidad sea del orden de $O(log_2(n))$. Aunque en el caso de la búsqueda interna este rendimiento suele ser más que aceptable, es posible que no lo sea en otras situaciones, como en la búsqueda externa, donde cada acceso tarda un tiempo significativo.

Una estrategia alternativa a la de buscar en la estructura es la de usar un método que nos diga directamente en qué posición está el elemento cuya clave proporcionamos.

Para poder usar esta estrategia necesitamos dos cosas. En primer lugar, una estructura que nos permita acceder directamente a cualquier posición, por lo que usaremos un array. La otra herramienta que necesitamos es una función que, dada una clave[1], nos devuelva directamente la posición donde debe guardarse el elemento con esa clave.

Funciones de dispersión

Esta función calculadora de direcciones debería tener varias características. Por ejemplo, debería ser fácil y rápida de calcular. También debería ser una función inyectiva, que quiere decir a claves distintas les debería asignar posiciones distintas dentro del array. También debería ser sobreyectiva, es decir, debería poder colocar alguna clave en todas las posiciones del array.

Este tipo de funciones se conocen como *funciones de dispersión*:

$$H : Clave \longrightarrow Posición$$

Una función de dispersión que cumpla las propiedades anteriores es una *función de dispersión perfecta*. Si disponemos de dicha función, los algoritmos de inserción, búsqueda y eliminación son triviales. En todos los casos basta con acceder a la posición que devuelve la función de dispersión.

Sin embargo, no es fácil encontrar una función de dispersión perfecta porque en la mayoría de los casos la situación que se da es que tenemos un conjunto de claves posibles muy grande del que sólo almacenaremos en la tabla un subconjunto pequeño, pero no sabemos a priori cuál es ese subconjunto. En esta situación, la función de dispersión tiene que asociar a varias claves la misma posición, con lo que deja de ser inyectiva.

Esta situación, en la que la función de dispersión relaciona varias claves con la misma posición es lo que se denomina *colisión*. Cuando la función de dispersión produce colisiones se complican notablemente los algoritmos porque es necesario resolver esa conflicto.

Ejemplos de funciones de dispersión

Selección de dígitos. Una función de dispersión basada en la selección de dígitos calcula la posición a partir de algunas de las cifras de la clave. Si, por ejemplo, a partir del DNI, seleccionamos la primera y la última cifra, tendríamos una posición en el rango [0..99]. Esta función, evidentemente, no es inyectiva, porque H(33242546) = H(300123876) = 36.

[1] nos basta con suponer que las claves son de tipo numérico. Si no es el caso, siempre es posible codificar la clave como un número al que se le aplica la función.

También puede producirse otro problema, dependiendo de la distribución de las claves. La primera cifra de los números de DNI suelen ser baja, menor que 5, con lo que, en la práctica, se están desperdiciando las posiciones superiores del array y se están agrupando las posiciones en la parte baja del array. Eso también ocurriría si la clave fuera el código postal y seleccionáramos las dos primeras cifras. Si los datos de entrada están agrupados por provincias, representadas por las dos primeras cifras, los elementos tenderán a agruparse alrededor de ciertas posiciones y dejarán libres otras, aumentando la posibilidad de que se produzcan colisiones. Por tanto, si la función de dispersión que se escoge no es inyectiva, al menos debe distribuir los elementos de la manera más uniforme posible para reducir el número de colisiones.

Plegado. Una solución que mejora la selección de dígitos es la de sumar los dígitos de la clave. En ese caso, H(33242546) = 29 y H(30123876) = 28. Estas funciones tampoco son inyectivas porque relacionan las diferentes permutaciones de una clave con una misma posición: H(33242546) = H(32324564) = 29. Otro inconveniente añadido es que pueden no ser sobreyectivas. En el ejemplo anterior la posición resultante está en el rango [0..72]. Se puede aumentar el rango sumando grupos de varias cifras. Si en nuestro ejemplo agrupamos las cifras de tres en tres y las sumamos, H(33242546) = 33 + 242 + 546 y H(30123876) = 30 + 123 + 876. En este caso, el rango de posibles posiciones ha aumentado a [0..2997].

Resto de la división entera. Estas funciones asocian a una clave la posición resultante del restor de la división entera con un número, que suele ser el tamaño del array. En ese caso, la función es sobreyectiva, pero no inyectiva. Si la función es H(cl) = cl `mod` 100, H(33242546) = H(25445246) = 46. Para lograr una distribución uniforme de las claves, se ha de escoger un número primo como denominador de la división.

Resolución de colisiones

Cuando intentemos insertar un elemento en una posición que ya ha sido asignada, no podemos dejar de insertar el elemento, porque el array puede no estar lleno. De hecho, en el caso extremo, sólo estaría el elemento que ha provocado la colisión.

Hay varias estrategias distintas a la hora de resolver una colisión. Unas se basan en buscan posiciones alternativas para el elemento y otras se basan en aumentar la capacidad de cada casilla del array.

Entre las primeras podemos encontrar:

Redispersión. En esta alternativa, se aplica una nueva función, la *función de redispersión*, a la posición generada por la función de dispersión. La función de redispersión, por tanto, es de la forma:

$$R : Posición \longrightarrow Posición$$

Cuando se va a insertar un elemento y se ha producido una colisión se intenta insertar en la posición dada por la función de redispersión. Si esa posición está libre, se inserta ahí el elemento. Si no lo está, se vuelve a aplicar a esa nueva posición la función de redispersión para lograr otra posición alternativa. El proceso acaba con éxito si se encuentra alguna posición libre. Si, tras aplicar repetidas veces la función de redispersión, llegamos a la posición original que generó la función de dispersión, no se puede incluir el elemento.

El algoritmo de búsqueda es similar. Inicialmente, se compara la clave a buscar con la que está en la posición dada por la función de dispersión. Si coinciden, hemos encontrado el elemento. Si no coiciden vamos aplicando la función de redispersión hasta encontrar la clave, en cuyo caso la búsqueda acaba positivamente o hasta que se produzca una de las dos siguientes circunstancias: que se encuentre una posición vacía o se vuelva a la posición inicial. En ambos casos, la búsqueda acaba sin encontrar el elemento.

Cuando consideramos el algoritmo de eliminación, se produce un caso que no habíamos tenido en cuenta. En principio, a la hora de eliminar un elemento con una clave dada, podemos seguir el proceso de búsqueda de la posición aplicando primero la función de dispersión y, las veces que haga falta, la función de redispersión. ¿Qué ocurre, sin embargo, si borramos ese elemento y marcamos la casilla como vacía? Hemos consideramo implícitamente en la discusión anterior que las casillas del array podían estar en dos estados, libres u ocupadas. Si durante el proceso de búsqueda de una clave llegamos a una posición de la que se ha borrado un elemento y

se ha marcado como libre, el algoritmo de búsqueda se detendría, siendo posible que el elemento buscado esté en otra posición y que no se haya insertado en esa porque estuviera ocupada cuando se insertó.

Veamos un ejemplo: tenemos un array cuyo índice es `[0..10]`, la función de dispersión es `H(cl) = cl mod 11`, y la de dispersión produce los siguientes resultados: `R(2) = 5`, `R(5) = 8`, `R(8) = 0`. Si queremos insertar un elemento con clave 24, `H(24) = 24 mod 11 = 2`. Como la posición 2 está vacía, insertamos ahí el elemento. A continuación, intentamos insertar el elemento con clave 46. `H(46) = 24 mod 11 = 2 = H(24)`, por lo que tenemos una colisión. Como `R(2) = 5`, intentamos insertar el elemento en la posición 5, que está vacía. Ahora intentamos insertar el elemento con clave 82. `H(46) = 24 mod 11 = 5`. La posición 5 está ocupada y, como `R(5) = 8`, insertamos el elemento en la posición 8. Si ahora buscamos la clave 82, se encuentra correctamente. Si borramos la clave 46, llegaremos a la posición 5. La marcamos como libre. Si en ese momento volvemos a buscar la clave 46, al buscar en la posición 5, vemos que está vacía y acaba el algoritmo de búsqueda sin encontrarlo.

Para remediarlo, cuando se borra un elemento, la casilla no se marca como vacía, sino como borrada. Cuando el algoritmo de inserción encuentra una casilla borrada puede insertar ahí el elemento. Cuando los algoritmos de búsqueda y de eliminación encuentran una casilla borrada, siguen buscando como si la casilla estuviera ocupada por otra clave.

Es importante que la función de redispersión sea sobreyectiva y que, a partir de cualquier posición, se visiten todas las otras posiciones del array antes de volver a la posición inicial. Una función de dispersión muy común es la de un incremento fijo en forma circular.

$$R(p) = (p + k)\ mod\ TAM\_ARR$$

donde `k` es una constante y `TAM_ARR` es el tamaño del array. Para que la función sea sobreyectiva, `k` y `TAM_ARR` deben ser primos entre sí. En el caso extremo, `k = 1`.

Doble dispersión. Un inconveniente propio de las funciones de redispersión es que se repiten todas las colisiones producidas por la función de dispersión, porque, al ser la función de redispersión independiente de la clave, produce la misma secuencia de posiciones alternativas para todas las claves que han colisionado. La doble dispersión consiste en aplicar una segunda función de dispersión a la clave. A partir de los valores dados, respectivamente, por las funciones de dispersión y de doble dispersión se calcula la siguiente posición, que debe ser diferente para claves distintas. Esto reduce la posibilidad de colisión, pero no garantiza que no las haya. Si tenemos como función de doble dispersión $H_{dd}(cl) = cl\ MOD\ 7$, y la función de redispersión se definiera como:

$$R(p, cl) = (p + H_{dd}(cl))\ mod\ TAM\_ARR$$

la clave 25 tendría como primera posición la 3 y luego iría buscando de 4 en 4 circularmente. Para 22, empezaríamos en la posición 0 e iríamos buscando de 1 en 1.

Cubetas. Esta solución consiste en ampliar la capacidad de cada casilla para que pueda almacenar más de un elemento. La estructura del array original se sustituye por un array bidimensional, cuyas casillas llamamos *cubetas*. Cuando se produce una colisión a la hora de insertar, se mira si hay espacio libre en esa posición. Si lo hay no hace falta buscar otra posición donde insertarlo. A la hora de buscar o de eliminar, hay que comparar la clave buscada con todas las claves que estén en esa posición.

Con esta solución se reduce la necesidad de buscar en otras posiciones, pero no se elimina. Si la cubeta de una posición está completamente llena, hay que repetir el proceso de búsqueda en posiciones alternativas. Este proceso, además, es más complicado porque hay que tener en cuenta el estado de ocupada, borrada o vacía para cada casilla de la cubeta. Si aumentamos mucho la capacidad de cada cubeta, el desperdicio de memoria será muy significativo.

Dispersión abierta. En esta solución se amplia la capacidad de cada casilla de manera dinámica. En vez de usar un array bidimensional, usamos una lista enlazada en cada casilla en la que habrá un nodo por cada elemento cuya clave tenga esa posición como resultado de la función de dispersión.

De esta manera eliminamos la necesidad de ir buscando los elementos en otras posiciones, porque podemos suponer que las listas enlazadas pueden ir creciendo indefinidamente. Sin embargo, cuando se producen muchas colisiones, el tamaño del las listas crece y el rendimiento en la búsqueda se degrada, porque la búsqueda de una clave en la lista enlazada es de coste lineas.

Reestructuración de la tabla de dispersión

Ls tablas de dispersión tienen la ventaja de su magnífico rendimiento cuando el número de colisiones es pequeño. Lamentablemente, cuando el índice de ocupación crece, también lo hace el número de colisiones y el rendimiento se degrada drásticamente, acercándose al orden lineal, muy por debajo del rendimiento ofrecido por otras implementaciones de estructuras de búsqueda.

La solución en estos casos es aumentar el tamaño de la tabla de dispersión para que caiga la relación entre las casillas ocupadas y las totales. Es decir, para que el rendimiento de la tabla de dispersión sea bueno, es necesario sacrificar algo de espacio de almacenamiento. Algunos autores recomiendan duplicar el tamaño del array cuando se llegue al 90 % de la ocupación en una estructura cerrada o el número de elementos sea el doble del número de casillas en una estructura abierta.

El aumento del tamaño de la tabla implica el cambio de la función de dispersión y, posiblemente, de las funciones de redispersión y doble dispersión.

Bibliografía

- Estructuras de datos y algoritmos / Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
- Intermediate problem solving and data structures : walls and mirrors / Paul Helman, Robert Veroff, Frank M. Carrano